

AD-A070 270    NAVAL INTELLIGENCE PROCESSING SYSTEMS SUPPORT ACTIVIT--ETC F/G 9/2  
STRUCTURED PROGRAMMING IN A DATABASE ENVIRONMENT, (U)  
MAY 79 L E TOWNER

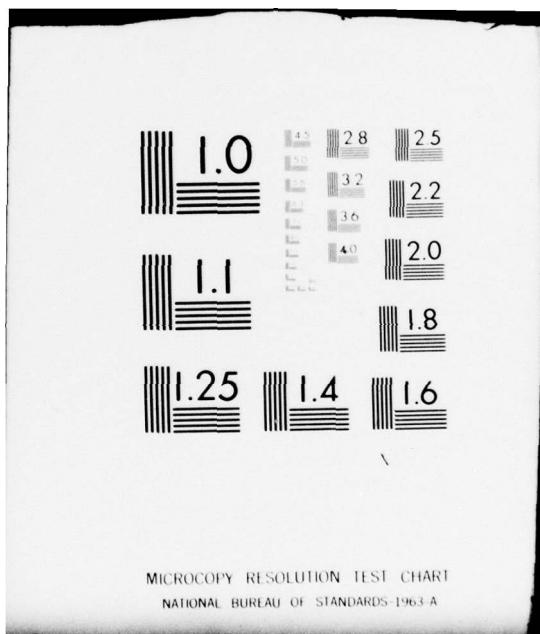
UNCLASSIFIED

NL

| OF |  
AD  
A070270



END  
DATE  
FILED  
7-79  
DDC



11  
LEVEL

12  
P

AD A070270

STRUCTURED PROGRAMMING IN A DATABASE ENVIRONMENT

10  
L. E. Towner

11  
24 May 79

12  
1981

Abstract

Structured programming has been actively endorsed by proponents of improved data processing management techniques for several years. Many examples of reduced costs and improved project control have been cited. Unfortunately, applications of these techniques have failed to meet the expectation in many cases.

The advent of the database management system (DBMS) as a major support tool in the data processing picture has also resulted in mixed results, often falling far short of the pre-installation build-up. Frequently this lack of DBMS achievement can be traced to inadequate control of the applications projects which will utilize the DBMS.

This paper describes the use of structured techniques, both analysis and programming, in overcoming the problems of database applications software support. Structured techniques are particularly applicable to the database environment because of the centralizing of procedural and logical programming functions around the DBMS software.

INTRODUCTION

Structured programming is one of those terms which has been tossed about in the data processing community for the last eight to ten years. It created some excitement and a great deal of controversy.

Structured programming has been viewed from two primary angles:

1. From management, who was looking (and still is) for a better and more effective way to control data processing costs. Management hoped that structured programming would provide a basis for accurate estimating of software costs and, even more important, improve the quality of the developed software.

This document has been approved  
for public release and sale; its  
distribution is unlimited.

DDC FILE COPY

DDC  
REF ID: A  
MAY 29 1979  
RECORDED  
C

19 05 24 040

410 508 100

2. From the programming staff, who frequently saw the advent of structured programming as a harnessing of their "creativity". Many programmers felt (and many have not changed their minds) that the structured programming approach was not worth the effort to learn and use.

Data base is another term which is currently being used by the industry with increasing frequency. During the past five years, the prospect of new ways of organizing data has created its share of excitement and controversy. The parallel between structured programming and data base is amazing.

As with structured programming, data base is being viewed from two angles:

1. From management, who is faced with the increasing complexity of business. Management must rely on computers more and more. Computers must have access to data to perform their job. The costs of managing data continues to grow. Management recognizes that data base may be their only hope to slow the escalating data management costs.
2. From the programming staff, who view generalized data base software with suspicion. Programmers are comfortable with software they know and understand. For many, this means software they have written. In the past, many of the commercial data base software packages were poorly documented and contained logic errors. Applications programmers were reluctant to use these packages because they could not fix the bugs or understand the documentation.

Structured programming, according to Ed Yourdon, one of best-known proponents of the technique, has lost its first round in the fight for acceptance. The round was lost because too much was said about the advantages of structured programming and not enough said about how it use it effectively. Programmers were turned off by "GO TO-less" rules and other factors which were poorly explained at best.

Data base faces a similar problem. There has been no difficulty convincing data processing management about the virtues of data base. It has even been relatively easy to convince company management that the cost of a data base package was worthwhile. Little effort, however, has been made to convince programmers that data base will really benefit them.

The similarities between structured programming and data base do not end with the visual pictures of management and programming staff. Functionally, both facilities provide tools and techniques which are complementary. When used together, they provide a powerful highly-effective data processing capability.

The remainder of this paper will describe the effective joining of structured programming and data base in an actual production environment. None of the techniques described are really new. They have been applied in such a way that the positive aspects of structured programming and data base are emphasized and many of the negative aspects are overcome.

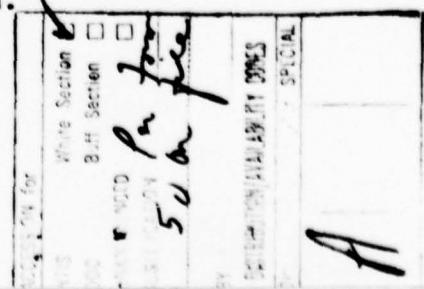
#### STRUCTURE PROGRAMMING FEATURES

There are many techniques and features which are gathered under the umbrella called structured programming. Some are more useful than others; some are, frankly, a deterrent to the acceptance and use of structured techniques. The features described below are those which were selected and implemented because of their particular match to the needs of data base development.

##### Modularity

Many programmers say they have written modular programs long before the structured approach was "discovered". True, but the manner in which modularity was practiced was quite different (in most cases) from that proposed by structured advocates.

The concept of a program, subroutine, or logic section having a single entry and exit point is important to clarity of program code. Where this approach is practiced, flexibility is improved and logic complexity (with its accompanying test time) is dramatically reduced.



A second aspect of modularity is the developing of libraries of common logic code. In the COBOL world, these "source library books" provide the ability to copy repeated logic into programs, saving coding, punching, errors in data preparation, and logic errors from omitting essential statements.

#### Top Down Design

The hierarchical structure, known also as the tree structure, has been used in business (organization charts, accounting, etc.) for hundreds of years. It is well known as an effective way of organizing a business or a chart of accounts. It is also an effective way of organizing software. Programs written which progress iteratively downward require little if any integration testing because the testing goes on all of the time. In turn, the flexibility to enhance the program is improved. This permits rapid response to changing requirement, the Achilles heel of most data processing installations

#### Maintainability

This feature is more of a by-product to structured programming rather than a primary feature. It is, however, a major factor in any software project. It has become as important to design for long-term maintainability of software as to meet today's user requirement.

Structured programs ARE more maintainable. They are, that is, if modularity and top down design techniques were followed and adequate comments were included in the program.

#### DATA BASE FEATURES

Database management systems come in all sizes, shapes, and flavors. No two are exactly alike, although those adhering to the CODASYL DBMS specifications appear alike on the surface. More about that later.

It is impractical, therefore, to define features of all DBMS packages which will assist the developer in improving the quality of the finished application while reducing costs over batch file systems. The list of features will be limited to a few major items which, if not present, will reduce the cost-effectiveness of a DBMS package in any environment.

### Integral Data Dictionary

A data dictionary is essential to effective use of a DBMS. Without this tool, the DBMS designer is handicapped by having to maintain database design data manually. When the data dictionary is specifically designed to support the DBMS, instead of one generally designed for multiple use, it should be possible to drive the schema definition software DIRECTLY from the dictionary. This dramatically reduces the design effort and duplication of definition material.

### Language Preprocessor

All DBMS products of a generalized form utilize call statements to communicate between the DBMS and the application program. The structure and complexity of these calls is frequently complex. A preprocessor which permits the programmer to use verbs, such as STORE, MODIFY, ERASE, instead of calls is an important item in potential error elimination and program self-documentation.

The preprocessor should also perform DBMS-associated housekeeping functions. The most important of these is to copy the definitions of the database records to be used into the program in the appropriate place. This feature enforces data naming conventions, insures consistent record definition, and reduces the volume of code prepared by the programmer.

### Data Independence

The DBMS must provide the capability to change the structure of the database itself without affecting the programs which do not require the results of the change. The subschema approach of the CODASYL specification typically provides this type of capability. The cost of future changes to the database can be significantly reduced with this feature.

### Full Network Data Structuring

Few non-CODASYL DBMS packages support this feature. Many potential users of DBMS are fooled into believing that this feature is unnecessary. It is possible to work around the problem. But this increases the design time and the programming effort goes up by an order of magnitude.

When the database structure is prepared which matches the actual interrelationship of the data, it is possible to reduce program complexity. The database has assumed, in effect, a portion of the program logic. It has been possible to write COBOL database programs where the Procedure Division is less than a page to accomplish functions which previously required three to four times the statements.

#### Physical Database Independence

Statistics, disputable of course, indicate that nearly one-half of conventional programming logic errors are somehow associated with I/O. The physical and structural attributes of a database should be transparent to the programmer. Whether the record is stored randomly or according to a sort order should be of minimum concern. Certainly, the programmer should be unconcerned where in the database the record is located.

#### STRUCTURING DBMS DATABASE DESIGNS

Just as structured programming is based on three logic building blocks, database structural relationships can be established using specific selected formats. CODASYL defines seven different data relationships which DBMS implementation should support. Indeed, those DBMS products which claim to conform to the CODASYL specification implement all seven. It is not necessary that all seven be used to develop effective database designs.

Figure 2 illustrates four basic building block structures for database development. Each has sufficient flexibility that, when combined such as shown in Figure 3, a completely networked data relationship results. The record building blocks are:

1. The BASE record, which is a primary entry point into the database. This record type is stored for direct retrieval.
2. The RELATIONAL record, whose job is to provide associations between other record types, usually BASE but occasionally INTERMEDIATE records. This record typically contains little substantive data.

3. The INTERMEDIATE record, called such because it is both a member and an owner of one or more other record types. This record nearly always contains substantive data.
4. The SUBSIDIARY record, which is always a member of some other record type. It always contains substantive data.

Developers of structured programming found that it was easier to create effective software when all logic was based on the Figure 1 forms. The same lesson has been applied successfully to database development.

At this point, it must be said that the technique is not limited only to CODASYL DBMS products. The technique itself is applicable to any DBMS. Like structured programming, however, the ease of use is greater with some products (in this case, CODASYL implementations) than others.

The advantages of structured database design include more than the visible initial investment savings. Long-term maintenance, frequently taking the form of changes to database structure and content, is reduced. Using already defined set structures, additional relationships can be added without forcing a restructuring of the physical database contents.

The application of structured techniques can be carried a further step. The composition of the database record itself can be designed to minimize the impact of change and to standardize the processing logic which supports the record. Figure 4 illustrates the general database record structure which is composed of four main parts:

1. The database pointers, controlled by the DBMS software, are fixed by the type of record. Predefining the possible relationships freezes the number of these pointers.
2. Fixed data fields, data elements which appear in such a large percentage of record types that they have been placed in all records. This part will be discussed in more detail in the next section.

3. Record identification fields, data elements which the DBMS and its users need to uniquely identify a record occurrence in the database. This group is used to define the random placement of BASE records in the database, and the sequence of placement of INTERMEDIATE and SUBORDINATE record occurrences.
4. The main data storage area, a fixed-length space which is converted to variable physical length for disk storing. This area will also be discussed in more detail. Analysis of repeating data requirements frequently identifies data elements which are heavily and commonly employed. The elements shown in Figure 5 will change under different organization requirements. The important aspect is that significant savings in software support can be achieved by always treating these fields identically. Some storage space waste can be expected where the fields are not required for individual record types but this is of less concern than the software costs necessary to handle individual cases.

The most frequent change to databases occurs within the makeup of record elements. Elements are added, deleted, or modified to satisfy changing organizational requirements. When such changes occur, they require corresponding changes to the physical database. Figure 6 illustrates an approach which eliminates the need to physically restructure the database while still satisfying the need for change.

A fixed general data area of 800 bytes is established for all record types in the database. Few record definitions, except those storing text, actually reach this size. Actual data usage, shown as n-bytes, is stored in the record. The remainder, 800-n, is logically present as far as the DBMS and application programs are concerned, but is physically compressed out of existence by DBMS data compression software. New elements simply reduce the 800-n value without affecting existing physical data storage.

The processing overhead required to compress and decompress physical records is, again, less costly than the effort required to periodically restructure the database, a cost which grows as the size of the database grows. It has been conservatively estimated that the actual computer time necessary to effect ONE database restructure would exceed the accumulated processing overhead for compression and decompression for six months of normal operation. Adding labor costs for restructuring clearly makes the technique cost-effective.

#### STRUCTURING DBMS APPLICATIONS SOFTWARE

The techniques addressed in the sections above lend themselves to developing highly modularized software. Common processing routines and functions are readily developed which, once implemented, are used repetitively throughout DBMS application development.

Figure 7 illustrates the logic used by input data processing software following this approach. Over 80 per cent of the average input processing program logic is common from one program to the next. The remaining 20 per cent deals with the validation of specific data elements and database access.

Even these functions may be modularized to reduce the amount of unique logic necessary. COBOL source library "books" have been developed which permit consistent verification of nearly all data conditions. Similar books take advantage of DBMS access ease to define standardized access processing.

Each of these highly structured programs is compiled as a separate processing entity, the implementation of another level of program structuring. Figures 8 and 9 illustrate the interaction of the input processing programs with batch and on-line program controllers. These controllers select the appropriate input program based on incoming data, creating a flexible transaction-driven database maintenance capability.

#### THE BOTTOM LINE

It is possible to achieve dramatic gains in productivity of software personnel through use of structured programming. Combining the features available in many DBMS products with structured techniques can result in even more significant productivity improvements.

The approaches described above are in use and have been successful beyond any expectations. Actual labor costs to develop input processing programs have been reduced by an average of 95 per cent. Calendar time to develop these same programs decreased by nearly 60 per cent.

The bottom line of data processing is the timely and effective support of the user community. Applying a combination of structured analysis and programming techniques with the supporting facilities of a good DBMS package may mean the difference between being a hero or unemployed. The end user, qualified or not, is the final judge.  
logoo

## BASIC STRUCTURED LOGIC BUILDING BLOCKS

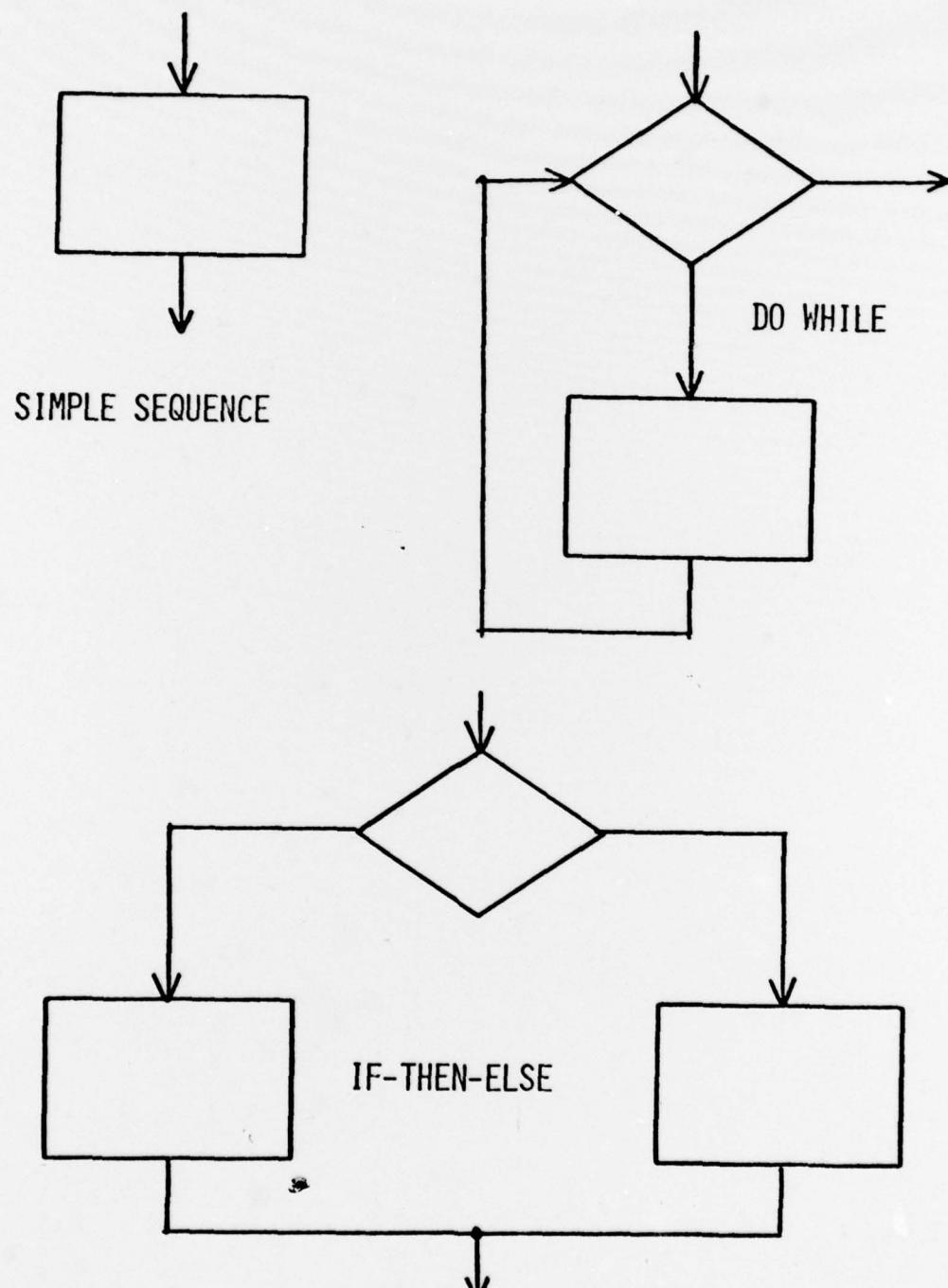


FIGURE 1

BASIC STRUCTURED DATABASE RECORD  
BUILDING BLOCKS

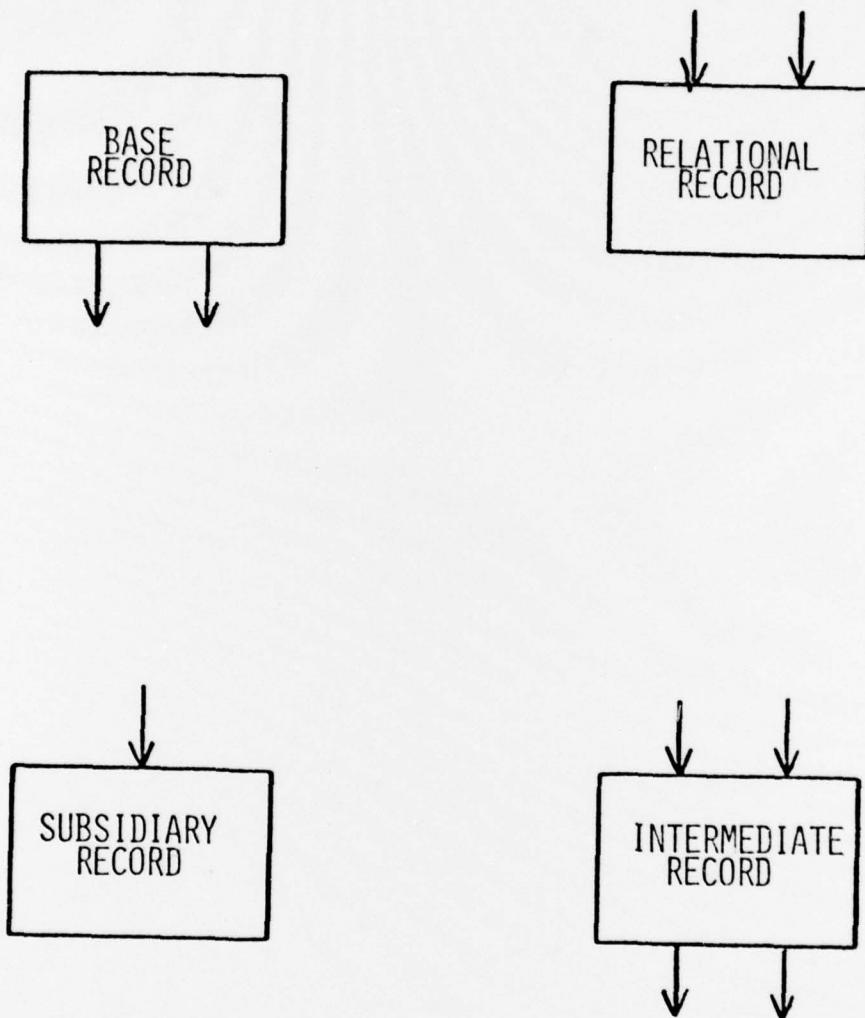


FIGURE 2

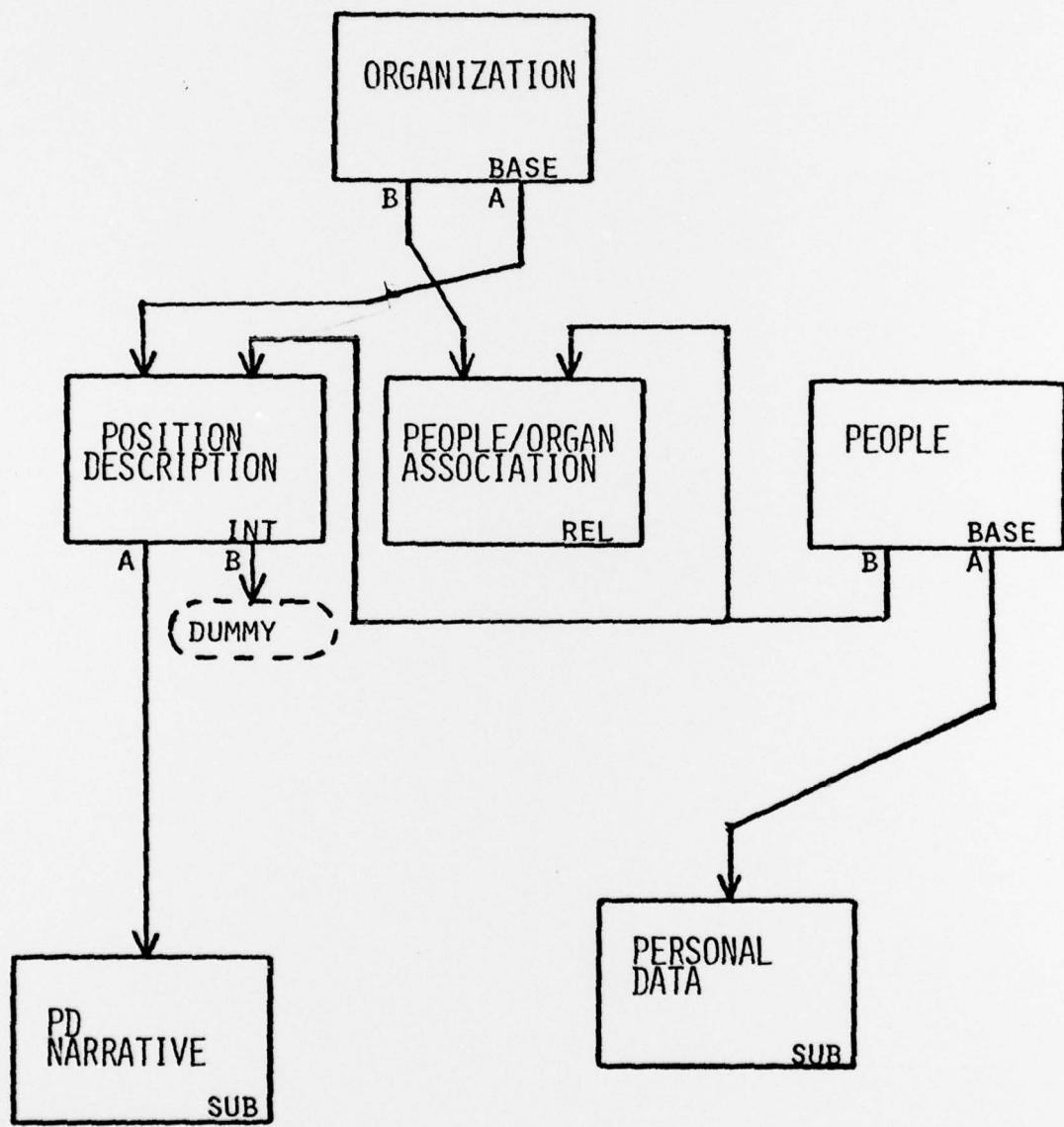


FIGURE 3

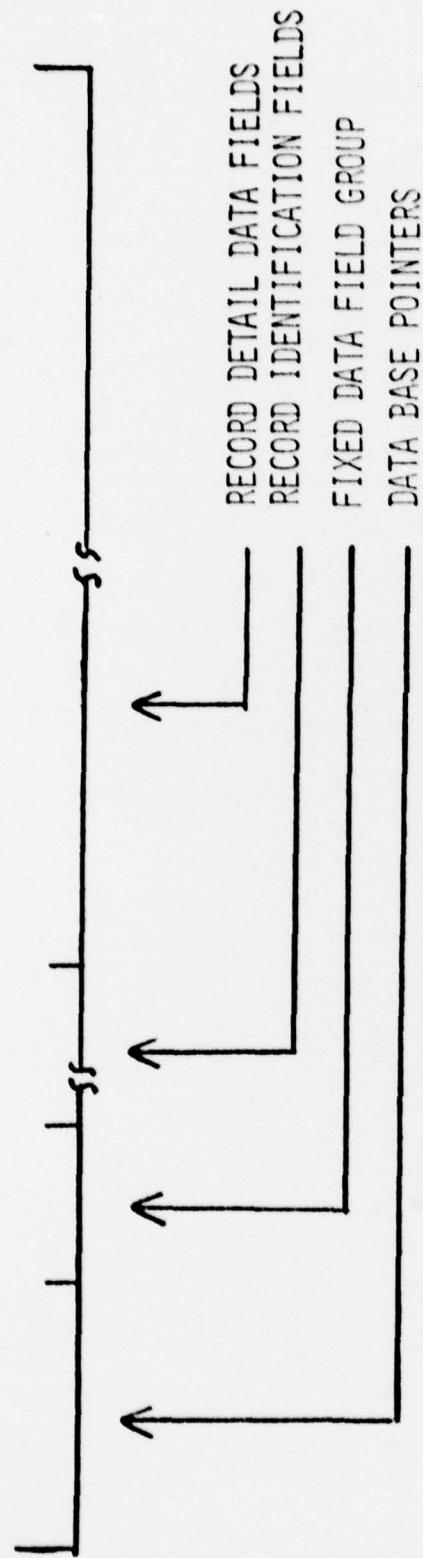


FIGURE 4

GENERAL DATABASE RECORD STRUCTURE

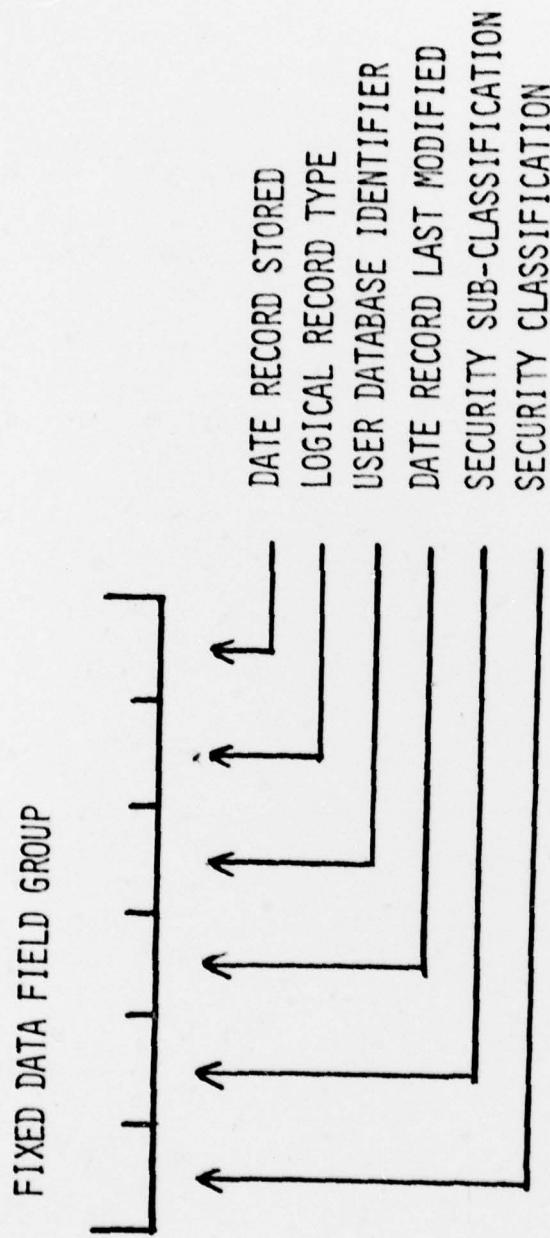


FIGURE 5

RECORD DETAIL DATA FIELD AREA

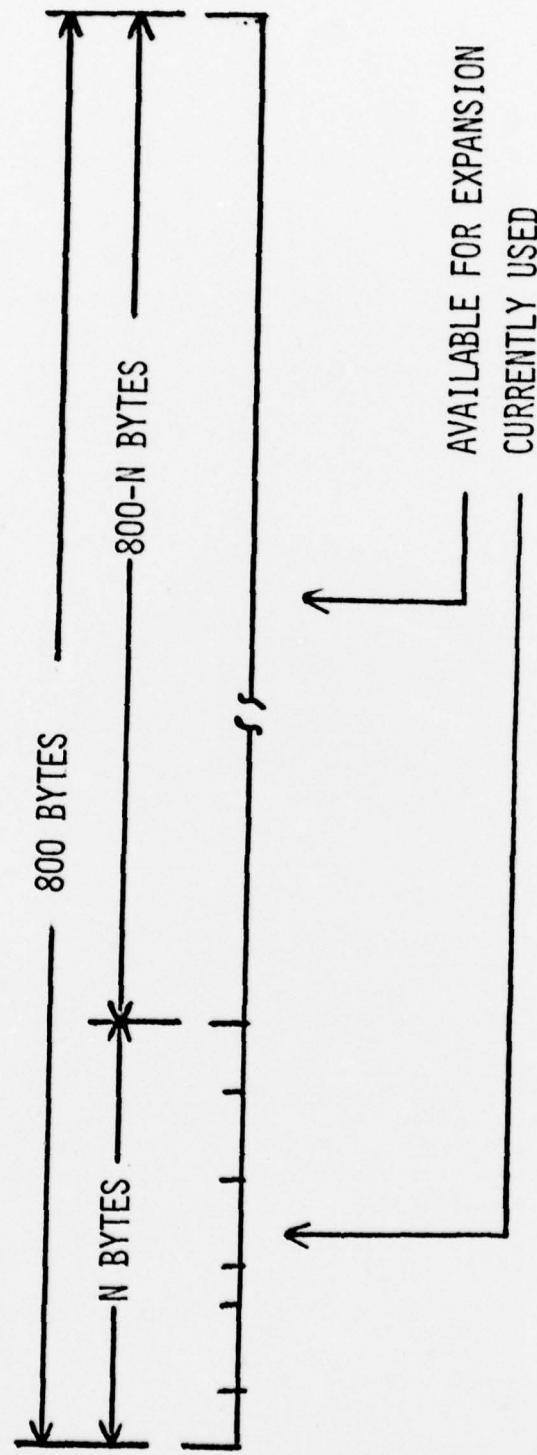


FIGURE 6

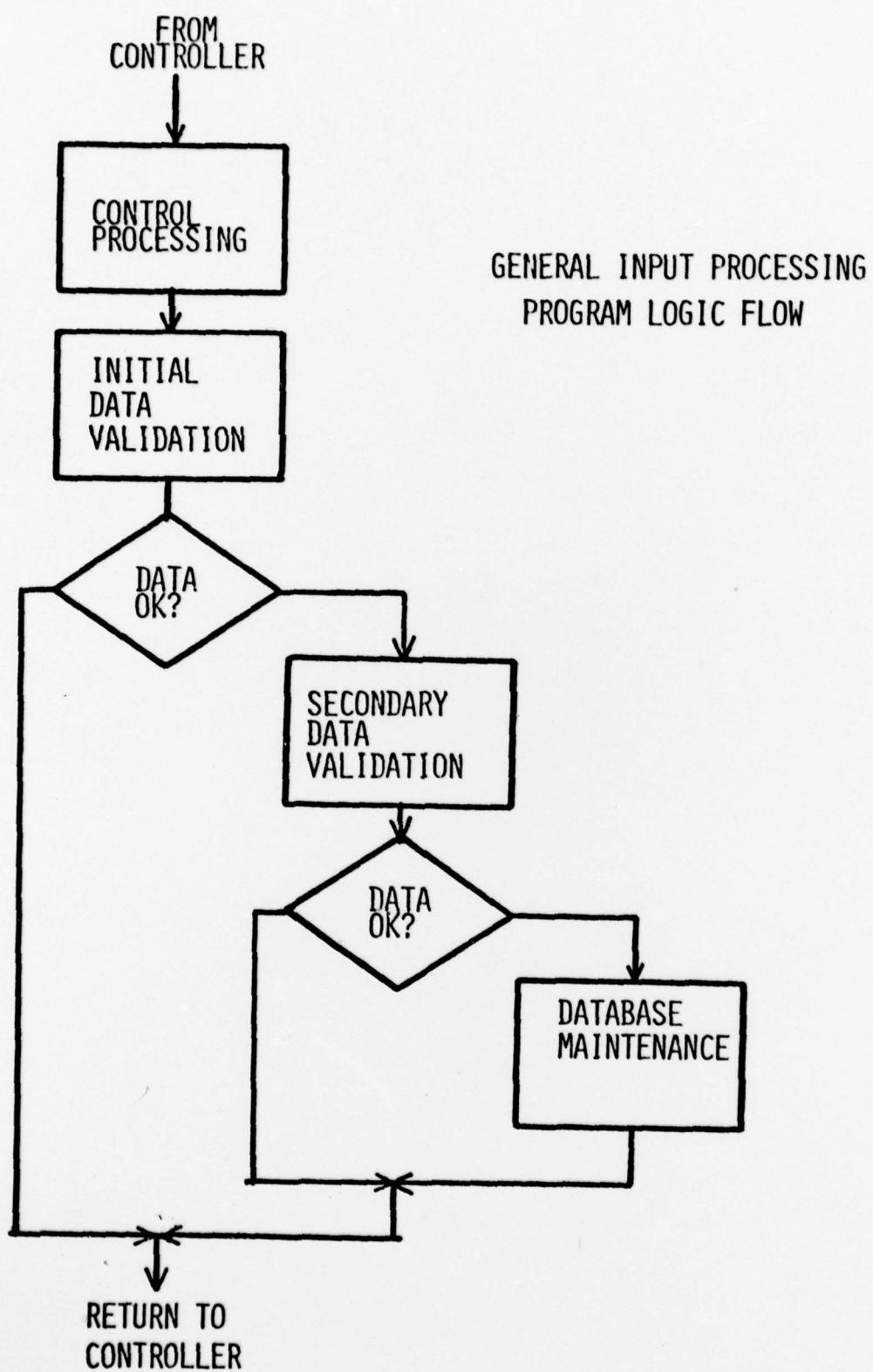


FIGURE 7

## CENTRALIZED BATCH INPUT PROCESSING

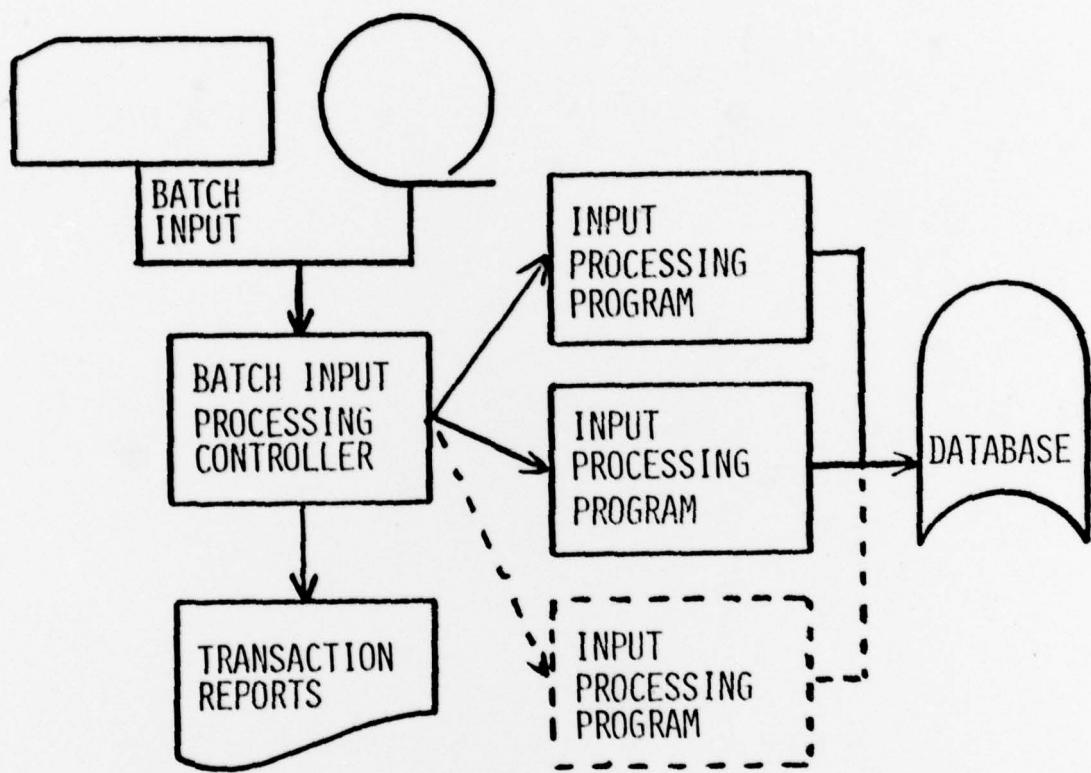


FIGURE 8

## CENTRALIZED ON-LINE INPUT PROCESSING

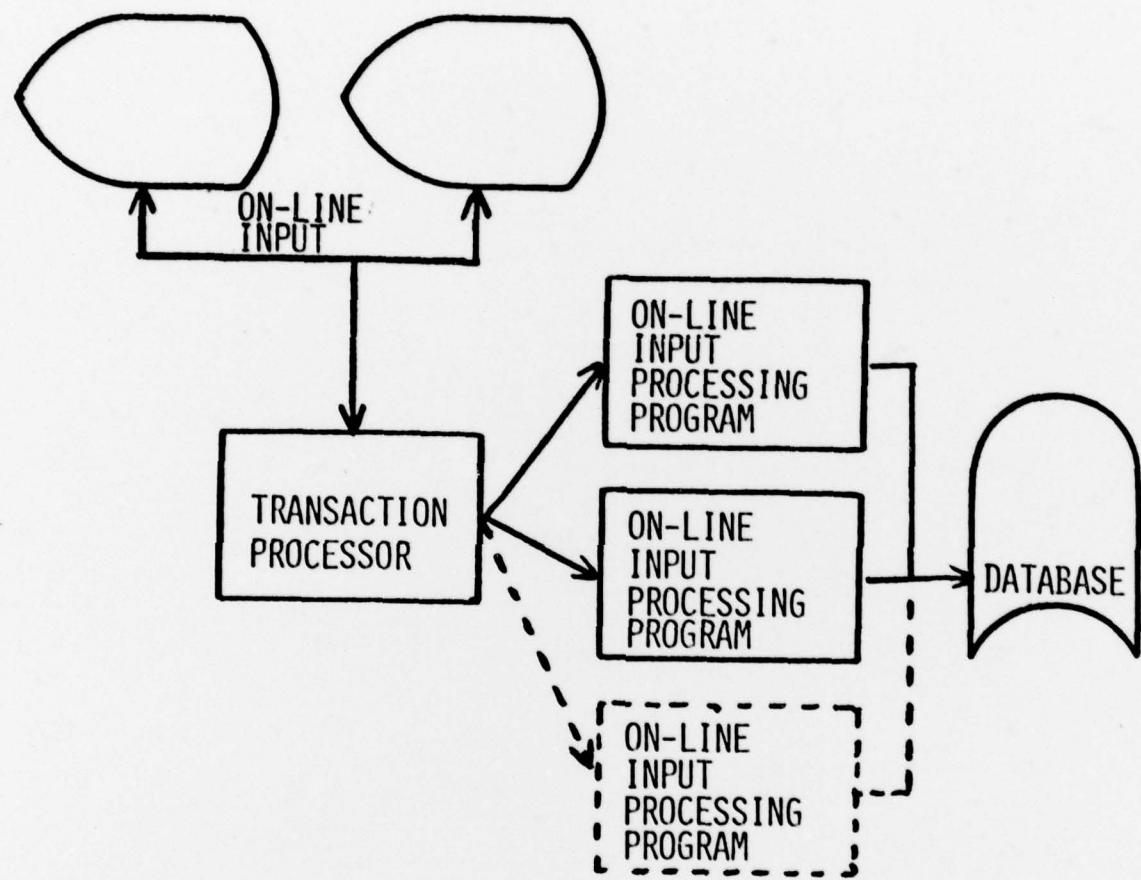


FIGURE 9